Proof of Necessary Work
Using Proof of Work to Verify State

Assimakis Kattis & Joseph Bonneau

New York University

May 9th 2019

A Problem of Size
Bitcoin Scaling Limitations

- Blockchain size increases linearly over time
- New clients require lots of brandwith & computation to join

Inefficient : All new clients need to do the *same* verification work to join the network from the beginning

Our Contributions
Proof of Necessary Work

Proof of Necessary Work : Use PoW to verify transactions

1. Allow light clients to verify state with minimal processing
2. Generate proofs ‘for free’ through PoW

Design Challenge
Proofs of State Validity

Important results from CS Theory :

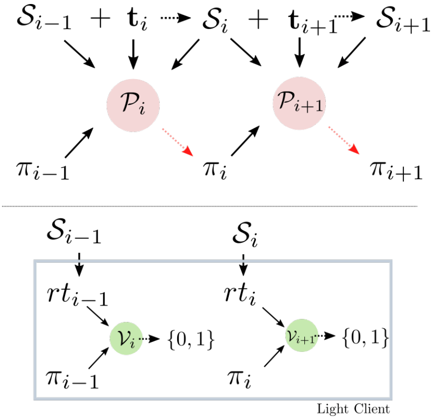
1. There exist 'small' proofs for *any* NP statement
2. Such proofs can verify previous proofs efficiently

Need proofs of state validity that :

1. can verify correctness of the *whole* chain
2. are small enough to add to the blockchain
3. can be checked with minimal resources

Idea : Use recursive SNARKs!

Proofs of State Validity
Succinct Blockchain Instantiation



Bitcoin naturally fits Incrementally Verifiable Computation

Prototype Design

Account-based prototype with simple payment functionality

Similar but *not* equivalent to Bitcoin :

1. No script or UTXOs
2. Doesn't support MULTISIG or arbitrary transaction types

Proofs of State Validity
Implementation Results

Succinct blockchain prototype results

# Tx	# Constraints	Generator $\mathcal{G}$		Prover $\mathcal{P}$		Verifier $\mathcal{V}$		pk Size (GB)	vk Size (kB)	π Size (B)
		Avg. (s)	σ (%)	Avg. (s)	σ (%)	Avg. (ms)	σ (%)			
1	441804	44.95	0.31	27.81	0.12	56.5	1.95	0.19	1.49	373
5	1561292	93.63	0.47	54.60	0.44	54.7	0.42	0.43		
10	2960652	143.57	0.70	88.09	0.30	54.8	0.60	0.75		
15	4360012	190.98	0.76	115.40	0.09	55.2	0.32	1.00		
20	5759372	234.65	0.79	140.93	0.15	55.2	0.57	1.29		
25	7158732	278.48	0.93	158.29	0.26	55.3	0.45	1.62		

TABLE 2. PROTOTYPE TIMES AND KEY SIZES FOR PREDICATES VERIFYING DIFFERENT NUMBERS OF TRANSACTIONS: AVERAGE RUNNING TIMES FOR SETUP $\mathcal{G}$, PROVER $\mathcal{P}$ AND VERIFIER $\mathcal{V}$ OVER 10 ITERATIONS ARE SHOWN ALONGSIDE PROVING/VERIFICATION KEY AND PROOF SIZES.

Benchmark : AWS ra5.2xlarge with 8 cores and 64GB of RAM

What did we achieve ?

Our prototype :

- produces block headers of size < 500 bytes for any number of transactions per block
- allows stateless clients to verify a block in < 60ms
- can achieve throughput of 100 tx/block using libsnark

Problem : The proofs take a long time to generate

Idea : Create them as part of the PoW process !

PoW from Proof Generation
Initial Approach

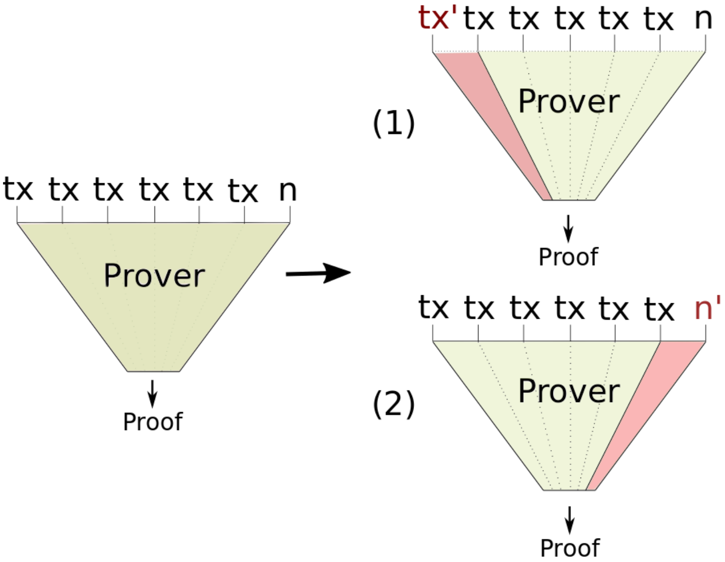
Generate π and accept if $\mathcal{H}(\pi) \leq d$, repeat otherwise

Need to add a random nonce to the proof every iteration

- Nonce is randomly sampled, changing π
- Probability of success is exponentially distributed

Problem : We can change n *without* recomputing all of π

Process favors returns to scale, leading to centralization !


tx tx tx tx tx tx n
Prover
Proof
(1)
tx' tx tx tx tx tx n
Prover
Proof
(2)
tx tx tx tx tx tx n'
Prover
Proof

Modelling Proof Generation

Need to ensure our predicate is 'hard' to solve in general

- We model this using a 'hardness' oracle $\mathcal{O}$
- $\mathcal{O}$ simulates hard computations used to generate π
- Prover has access to $\mathcal{O}$ but can reuse previous information

In current succinct SNARK implementations, $\mathcal{O}$ provides access to *modular exponentiation* in some group $\mathbb{G}$

Reduces to hardness in the Generic Group Model (GGM)!

Formalizing the Model

Definition (ϵ-Hardness)

For $\ell \in \text{poly}(\lambda)$ and length λ inputs, $f^{\mathcal{O}}$ is ϵ-hard if $\forall \mathcal{A}$ performing less than $(1-\epsilon)N\ell$ queries to oracle $\mathcal{O}$, where N number of queries required for one evaluation of $f^{\mathcal{O}}$, the following is negligible in λ :

$$\Pr\left[\begin{array}{c} \forall i \in [\ell], \pi_i = f^{\mathcal{O}}(a_i) \\ \forall i, j \in [\ell], a_i \neq a_j \Leftrightarrow i \neq j \end{array} \middle| \; \{\pi_i, a_i\}_{i=1}^{\ell} \leftarrow \mathcal{A}(1^{\lambda}) \right]$$

Intuition : A large prover only gets an ϵ advantage from previous computation when generating proofs

Committing to the Nonce

Leveling the Playing Field

We hope to solve this by committing to the nonce in the proof

- Valid blocks now a (sensitive) function of n
- Changing any input leads to an invalid configuration
- Prevents previous proofs from inducing speedups

Computing proofs with random n prevents returns to scale

Result : Miners have to compute the whole proof

Adding Nonce to State
Altering Merkle Computations

Account-based models keep state in a Merkle tree :

1. Checks old Merkle paths
2. Computes new Merkle paths
3. Checks that signature and amounts are valid

Idea #1 : Link state and nonce through a 'seed' parameter : $\rho = \mathcal{H}(n|state)$. Requires only one verification of a PRF $\mathcal{H}$

Result : Altering any part of the input means a new valid ρ is required, which is unpredictable by the security of $\mathcal{H}$

Creating Hard Predicates

State verification happens *without* the seed. Most computation ($\sim$ 97%) in current account-based models verify Merkle paths

Problem : This only requires access to state! An adversary can reuse work as ρ doesn't alter the vast majority of computation

Goal : Alter predicate to embed ρ in the verification process

Strawman : Insert n in every *updated* leaf. New Merkle paths then change unpredictably as a function of the nonce

If we could inject our nonce in *all* Merkle paths, would be done

Problem : We only alter half of them! Gives an $\epsilon \approx 1/2$

New Idea : Modify hash function by 'cloaking' it with ρ

Design Challenge : Modify our hash function to use ρ 'almost everywhere', while outputting the same result as before

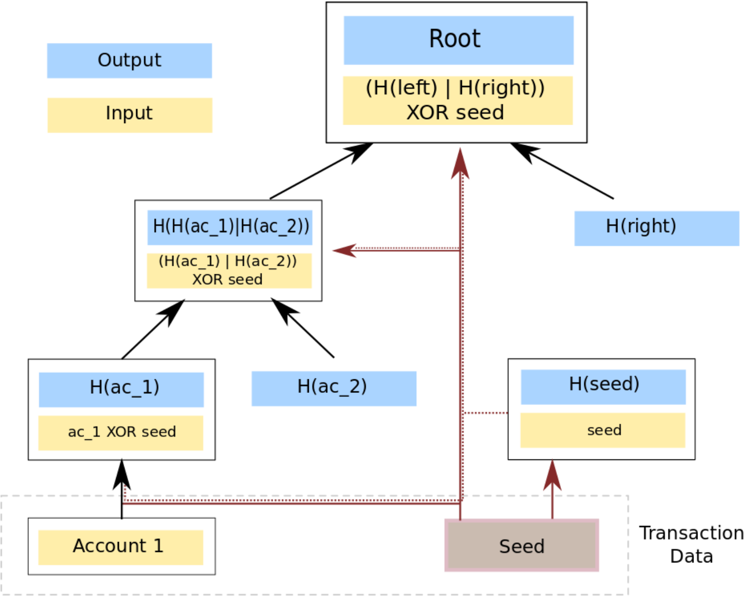
Cloaking the Pedersen Hash

For generators $\{g_i\}_{i=1}^{n}$ in $\mathbb{Z}_p^*$, an n-bit Pedersen hash is :

$$\mathcal{H}(x) = \prod_{i=1}^{n} g_i^{x_i} \text{ where } x_i \text{ the } i\text{-th bit of } x$$

Idea # 2 : Compute $\mathcal{H}(\rho)$ and then calculate $\mathcal{H}^{'}(x \oplus \rho, \mathcal{H}(\rho))$
Can compute $\mathcal{H}(\rho)$ *once* per block. We can then use this with a transformation $\mathcal{H}^{'}$ for which $\mathcal{H}^{'}(x \oplus \rho, , \mathcal{H}(\rho)) = \mathcal{H}(x)$ for any x

Efficiency : Need to verify the XOR input. Adds $O(n)$ overhead for a $\sim 20\%$ increase in proving time per $\mathcal{H}$ circuit


Output
Input
Root
(H(left) | H(right))
XOR seed
H(H(ac_1)|H(ac_2))
(H(ac_1) | H(ac_2))
XOR seed
H(right)
H(ac_1)
ac_1 XOR seed
H(ac_2)
H(seed)
seed
Account 1
Seed
Transaction
Data

Putting it all together

We demonstrate how to cloak predicates with a nonce n, making information reuse impossible

$\epsilon \approx 3\%$ with overhead $\sim 20\%$ per block in our predicate when implementing the previous ideas w/o optimization

For our 20 tx predicate, this means $\epsilon \geq 0.3\%$ if using SHA

$\epsilon \geq |C_{\mathsf{PRF}}|/|C_{\mathsf{Block}}|$ becomes our lower-bound

Proof Chains

Improving System Throughput

Discarding previous proofs is also wasteful - can we do better ?

We propose 'Proof Chains', an extension to PoNW that :

- requires miners to build *on top* of previous proofs
- submits all proofs in the chain when difficulty is satisfied

Result : Throughput increase 'for free' in our implementation

Related Work

An ideal proof system would :

1. require verifier *succinctness* (for efficient IVC),
2. also be *trustless* (no trusted setup),
3. and *quantum-resistant*.

Recent work is rapidly approaching these capabilities

Since our modifications are on the *predicate layer*, such improvements are complementary to our approach

Remark : Our design uses IVC as a *black box*. Can switch in *any* proof system that does IVC with the same guarantees

Future Work

We identify various areas for future work :

- Generalize to arbitrary proof systems
- Design cloaking properties for other (faster) hash functions
- Extend to full Bitcoin functionality (soft fork ?)

Contact : kattis@cs.nyu.edu